Comunicación intercomponente en *ArchStudio 3.0*: diseño del conector "message filtering"*

Isabel Muñoz Fernández[1], Jorge E. Pérez Martínez[1], Enrique Soriano Salvador[2]

[1] Departamento de Informática Aplicada, Universidad Politécnica de Madrid
Ctra. de Valencia, Km. 7, 28031 Madrid, Spain
{imunoz,jeperez}@eui.upm.es
[2] Grupo de Sistemas y Comunicaciones, ESCET, Universidad Rey Juan Carlos
C/ Tulipán s/n, 28933 Móstoles (Madrid), Spain
{esoriano}@gsyc.escet.urjc.es

Resumen. Tradicionalmente, el foco de atención en el desarrollo de una arquitectura software se ha centrado en los componentes, relegando a un segundo plano las formas de interacción entre estos componentes: los conectores. Sin embargo, para que un sistema funcione correctamente es necesario dedicar tanta atención a los conectores como a los componentes. En este trabajo presentamos un estudio sobre la herramienta *ArchStudio 3.0*. El análisis se ha centrado en las capacidades de dicha herramienta para soportar la comunicación entre componentes mediante paso de mensajes. Sobre dicha herramienta se han realizado correcciones en el código, se han rediseñado algunos de sus elementos para mejorar la eficiencia y se ha diseñado e implementado la política de filtrado C2 conocida como *message filtering*.

1. Introducción

En el marco del proyecto CICYT (TIC: 2001-1586-C03-01), estamos estudiando la definición de una arquitectura de componentes distribuidos para tolerancia a fallos. Tras varias aproximaciones hemos concluido con un modelo independiente de la plataforma (PIM) de dicha arquitectura descrita en el estilo arquitectónico C2 [4]. El siguiente paso, siguiendo la metodología MDA [5], consiste en la transformación de dicho modelo en un modelo específico de la plataforma (PSM) tomando, en este caso, a Java como plataforma destino. La elaboración de dicho PSM implica: 1) cómo caracterizar un componente C2 en Java, y 2) cómo soportar el mecanismo de comunicación definido en C2 con las herramientas de las que dispone Java. Actualmente estamos centrados en este último aspecto: la comunicación entre componentes C2 utilizando Java. Para ello hemos analizado, por un lado, las distintas tecnologías disponibles en Java para el desarrollo basado en componentes [7] y, por otro lado, *ArchStudio 3.0*.

* Parcialmente financiado por el MCyT (TIC 2001-1586-C03-01). Deseamos agradecer a Antonio Fernández Anta y a Sergio Arévalo Viñuales su supervisión y aportaciones.

En este trabajo se describe un análisis de las capacidades de *ArchStudio 3.0* para soportar la comunicación entre componentes. *ArchStudio 3.0* es un entorno desarrollado por *The Institute for Software Research at the University of California, Irvine* [3] orientado a soportar arquitecturas software elaboradas con el estilo arquitectónico C2 (entre otros) e implementadas en el lenguaje Java. Dicho estudio ha concluido con una serie de modificaciones a *ArchStudio 3.0* así como con la inclusión de la política de filtrado *message filtering*, definida en C2 y que no estaba soportada por ese entorno.

La organización de este trabajo es la siguiente. En la sección 2 se definen los conceptos de arquitectura software y de estilo arquitectónico para establecer un marco conceptual de referencia. La sección 3 describe resumidamente el estilo arquitectónico C2. En la sección 4 se analiza *ArchStudio 3.0* enfocando el estudio en las formas de interacción entre componentes que soporta dicha herramienta. La sección 5 describe nuestras propuestas para mejorar y ampliar *ArchStudio 3.0*. Por último, la sección 6 presenta algunas conclusiones de este trabajo y futuras líneas de investigación.

2. Arquitectura software

Sobre el concepto de arquitectura software, Shaw y Garlan [6] indican: "Brevemente, la arquitectura software implica la descripción de los elementos desde los que se construyen los sistemas, las interacciones entre esos elementos, patrones que guían su composición y restricciones en esos patrones". Por otro lado, Bass, Clements y Kazman [1] indican que: "un estilo arquitectónico es una descripción de tipos de componentes y un patrón de transferencia de datos y/o control en tiempo de ejecución". Y más concretamente, estos autores indican que un estilo arquitectónico está determinado por: (1) un conjunto de tipos de componentes; (2) una distribución topológica de esos componentes; (3) un conjunto de restricciones semánticas; y (4) un conjunto de conectores que median la comunicación, coordinación o cooperación entre componentes.

3. El estilo arquitectónico C2

C2 es un estilo arquitectónico que se puede definir como una red de componentes concurrentes unidos por dispositivos de encaminamiento de mensajes [4]. Este estilo se caracteriza por:

- el principio de la visibilidad limitada o independencia del substrato,
- el *Multthreading*: un componente puede tener su(s) propio(s) flujo(s) de control,
- la no suposición de la existencia de espacio de direccionamiento compartido,
- la invocación implícita y
- la comunicación mediante paso asíncrono de mensajes.

Los elementos claves de una arquitectura C2 son los componentes y los conectores. Ambos tienen definidos un domino *top* y un dominio *bottom*. El *top* de un componente puede estar conectado al *bottom* de un único conector. El *bottom* de un componente puede estar conectado al *top* de un único conector. Un conector puede estar conectado con cualquier número de componentes y/o conectores. Los componentes sólo se pueden comunicar a través de los conectores; la comunicación directa entre componentes está prohibida. Cuando se conectan juntos dos conectores debe ser desde el *bottom* de uno al *top* del otro. Los componentes se comunican por paso de mensajes; los anuncios (*notifications*) atraviesan la arquitectura hacia abajo; las peticiones (*requests*), hacia arriba. Los conectores son los responsables de encaminar los mensajes y potencialmente hacer multidifusión (*multicast*) de los mismos. También son los responsables de soportar las políticas de filtrado definidas en C2. En la figura 1 se describe (con un PIM) parte de una arquitectura de componentes distribuidos para tolerancia a fallos en el estilo C2.

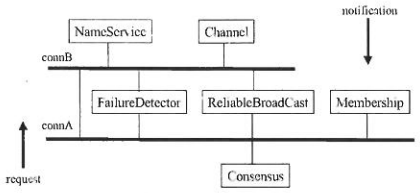

Fig. 1. Un ejemplo de arquitectura en estilo C2

Los conectores ligan componentes y son los responsables de encaminar los mensajes mediante el uso de alguna de las siguientes políticas de filtrado: (1) *no filtering*, cada mensaje se envía a todos los componentes conectados en el dominio correspondiente del conector (*bottom* para *notifications* y *top* para *requests*); (2) *notification filtering*, cada notificación se envía sólo a aquellos componentes que se hayan registrado por ella; (3) *message filtering*, cada mensaje se envía sólo a aquellos componentes que pueden comprenderlo y responderlo; (4) *prioritized*, cada componente tiene una prioridad asignada, de tal forma que el conector envía una *notification* a cada componente, al que está conectado, en el orden de prioridad establecida y hasta que se cumpla una cierta condición; (5) *message sink*: el conector ignora todos los mensajes que le llegan.

4. Comunicación entre componentes en *ArchStudio 3.0*

ArchStudio 3.0 está organizado en un conjunto de paquetes que contienen las aproximadamente 16000 líneas de código Java (escasamente documentadas) de las que se compone. El paquete más importante es `c2.fw`, ya que permite la definición de los elementos de una arquitectura y de la máquina virtual de *ArchStudio 3.0* para ejecutar dicha arquitectura. Otros paquetes complementarios a destacar son: (1) el paquete `c2.pcwrap`, que permite recubrir el mecanismo de paso de mensajes

mediante interfaces procedurales; (2) el paquete `c2.conn.fred`, que permite interconectar arquitecturas residentes en máquinas distintas; y (3), el paquete `c2.legacy`, que especializa el núcleo de `c2.fw` para la semántica del estilo C2.

En las siguientes secciones se estudiarán las dos semánticas de comunicación que *ArchStudio 3.0* ofrece a los elementos de una arquitectura: la comunicación mediante paso de mensajes y la comunicación mediante llamadas a procedimientos remotos. Previamente se describirá el diseño interno de la máquina virtual de *ArchStudio 3.0.*

4.1. Visión conceptual de una arquitectura software en *ArchStudio 3.0*

Una arquitectura en *ArchStudio 3.0* está formada por la interconexión jerárquica de un conjunto de elementos de computación o *bricks*, que cuentan con espacios de direccionamiento disjuntos, se ejecutan concurrentemente y se comunican mediante paso de mensajes. Un *brick* se compone de dos partes bien diferenciadas (fig. 2): la parte del comportamiento y la parte de comunicación con el resto de la arquitectura. Ambas, pueden especializarse para que el *brick* describa un estilo arquitectónico concreto; por ejemplo, en C2 un *brick* se comportará como un componente o como un conector. Cada *brick* cuenta con un dominio *top* y un dominio *bottom*, ambos formados por dos puertos: *in* y *out*. Por el puerto *top.in* recibirá mensajes de los puertos *bottom.out* de los *bricks* conectados con él por la parte superior de la arquitectura, mientras que por el *top.out* enviará mensajes a los puertos *bottom.in* de dichos *bricks*. El funcionamiento del dominio *bottom* es simétrico al dominio *top*.

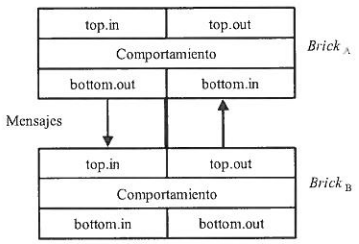

Fig. 2. Comunicación entre dos *bricks*

La interfaz `c2.fw.Brick` permite definir los *bricks* de una arquitectura software, mediante métodos que soportan sus tres funciones básicas: el control de su ciclo de vida, el estado de los *threads* internos al *brick* y el mecanismo del paso de mensajes. De esta interfaz se destacan los siguientes métodos relacionados con la comunicación:

```
public void handle(Message m);
public void send (Message m, Interface iface);
public void addMessageListener (MessageListener l);
```

Cada vez que el *thread* de control del *brick* reciba un mensaje, llamará al método `handle` para que éste interprete el mensaje que ha recibido. Por su parte, para enviar un mensaje, el *thread* invocará el método `send` que delega la copia del mensaje al

gestor de mensajes `MessageHandler` (de la máquina virtual de *ArchStudio 3.0*) mediante un mecanismo de eventos. Así, el método `addMessageListener` permite que el `MessageHandler` se subscriba al evento "envío de mensaje" generado por un *brick* cuando éste invoca la operación `send`.

Por otra parte, si el *brick* recibe mensajes de múltiples tipos, la programación del método `handle` para tratar todos los casos sería farragosa. Por esta razón, la interfaz `c2.fw.DelegateBrick` extiende a `c2.fw.Brick` para facilitar el tratamiento especializado de los mensajes, ofreciendo el método `addMessageProcessor` que permite crear en el *brick* tantos procesadores de mensajes `MessageProcessors` como tipos de mensajes se quieran tratar.

4.2. La máquina virtual de *ArchStudio 3.0*

ArchStudio 3.0 ofrece una máquina virtual cuya interfaz está formada por servicios orientados al manteniendo de la topología, a la gestión del paso de mensajes, a la gestión del ciclo de vida de los *bricks* y al control de sus *threads* creados. Esta interfaz está soportada por el controlador del sistema o `ArchitectureController`, que distribuye el control de los aspectos anteriores en los siguientes gestores (fig. 3):

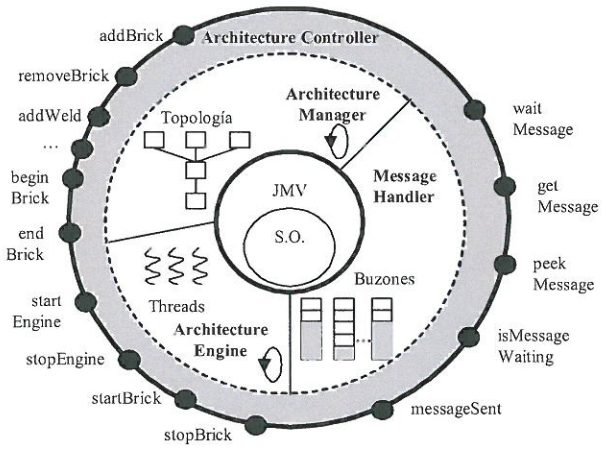

Fig. 3. La máquina virtual de *ArchStudio 3.0*

- **el gestor de la topología** o **`c2.fw.ArchitectureManager`**, es un *thread* que controla la misma, ofreciendo operaciones tales como añadir o eliminar un *brick* y establecer o eliminar un enlace (*weld*) entre *bricks*,
- **el gestor de paso de mensajes** o **`c2.fw.MessageHandler`**, que soporta los servicios para la comunicación entre los puertos de los *bricks*. Este gestor ofrece dos implementaciones recogidas en las siguientes clases reactivas:

`c2.fw.SingleQueueMessageHandler` que utiliza una única cola de mensajes para todos los *bricks* de la arquitectura del sistema, y `c2.fw.OneQueuePerIntefaceMessageHandler` que reserva un buzón FIFO para cada puerto de entrada (*top.in* y *bottom.in*) de un *brick* y es la que se utiliza por omisión. En ambas clases, la comunicación asíncrona se consigue con buzones con capacidad mayor que cero.

- **el motor de la arquitectura** o **`c2.fw.ArchitectureEngine`**, es un *thread* que soporta la política de *threading* elegida por el usuario para los *bricks* del sistema y el control del ciclo de vida de los mismos. También, existen varias políticas de *threading* disponibles recogidas en las siguientes clases:
  - `c2.fw.ThreadPoolArchitectureEngine` que permite que los *threads* se compartan entre todos los *bricks* del sistema;
  - `c2.fw.OneThreadSteppableArchitectureEngine`, para ejecutar paso a paso el motor y,
  - `c2.fw.OneThreadPerBrickArchitectureEngine` que asigna un *thread* de control para cada *brick*. Esta última es la política por omisión.

4.3 El sistema de comunicación: la interfaz `MessageHandler`

La interfaz `c2.fw.MessageHandler` soporta un sistema de comunicación directo y asíncrono de paso de mensajes entre *bricks*, puesto que es un requisito de la comunicación en C2. Los métodos más significativos se describen a continuación:

```
public Message waitMessage(BrickIfaceIdPair endpoint)
                           throws InterruptedException;
public Message getMessage(BrickIfaceIdPair idPair);
public void messageSent(Message m);
```

Ya que un mensaje va dirigido a un puerto *in* de un *brick* destino, la comunicación debe ser directa. Por tanto, en todos los métodos, el destinatario se indica mediante el par: identificador del *brick* y dominio *top* o *bottom* de dicho *brick*, recogido en el tipo `BrickIfaceIdPair`. La operación `waitMessage` bloquea al *thread* llamante si no hay un mensaje disponible en el puerto y *brick* especificados. Si hay un mensaje, se saca del buzón y el *thread* continua. El método `getMessage` tiene semántica de recepción no bloqueante, de tal forma que si no hay un mensaje disponible se devuelve `null` y el *thread* no se bloquea. En caso contrario se le devuelve un mensaje. Finalmente, la operación `messageSent`, utiliza el sistema de eventos que da soporte al envío de mensajes. En este sistema intervienen todos los *bricks* del sistema y el gestor de mensajes. El gestor de mensajes (`MessageHandler`, ver fig. 3) se suscribe al evento "envío de mensaje" invocando el método `addMessageListener` de cada *brick* del sistema. Un *brick* genera el evento "envío de mensaje" cuando invoca su método local `send` (ver apartado 4.1), que dispara la ejecución del método `messageSent` del gestor de mensajes. Dicho método copiará el mensaje desde el espacio de direccionamiento del *brick* emisor al buzón destino del *brick* receptor.

4.4 Llamadas a procedimientos mediante paso de mensajes

El paquete `c2.pcwrap` (*procedure call wrapping*) permite que dos *bricks* se comuniquen mediante una interfaz procedural que esconda el mecanismo de paso de mensajes. El modelo de comunicación que se sigue es el típico modelo cliente/servidor: un *brick* servidor ofrece un servicio, definido por su interfaz, al resto de los *bricks*; los *bricks* clientes importan dicho servicio para poder usarlo. La invocación a un método remoto sigue la semántica definida por Birrell y Nelson en [2]. Así, la semántica de la llamada al método remoto intentará parecerse lo más posible a la semántica de una llamada a un método local, puesto que las llamadas serán síncronas y los errores de ejecución que se produzcan en el servidor remoto se propagarán al *brick* local. La clase principal dentro del paquete `c2.pcwrap` es la clase `c2.pcwrap.EBIWrapperUtils` cuyos métodos más importantes son:

```
public static void deployService(DelegateBrick b,
  Interface callIface, Object api, Class[] interfaces);

public static Object addExternalService(
  DelegateBrick b,interface iface, Class serviceClass);
```

El método `deployService` permite que un *brick* exporte un servicio. El parámetro `b` indicará el *brick* que va a exportar el servicio; `callIface` indicará el puerto del *brick* por dónde se atenderán las peticiones. El parámetro `api` se refiere al objeto dentro del componente que va a soportar los servicios que se describen en el parámetro `interfaces`. Finalmente, `interfaces` contiene las interfaces Java compiladas de cada servicio a exportar.

El método `addExternalService` permite que un brick use un servicio. El primer parámetro `b`, indica qué *brick* será el cliente de dicho servicio; `iface` indica el puerto por el que se generarán las peticiones y, por último, `serviceClass` es la interfaz del servicio que el *brick* `b` pretende usar. Como resultado de la invocación de este método se devuelve un *proxy* del servicio que el *brick* usará para realizar llamadas a los métodos de dicho servicio. Por ejemplo, si el *brick* B_A quiere usar un servicio `S` por su interfaz *top*, obtendrá su *proxy* invocando:

```
proxyS = EBIWrapperUtils.addExternalService(B_A,'top',
         S.class)
```

Donde `S.class` es la interfaz del servicio `S`, y `proxyS` es el *proxy* del servicio `S` para el *brick* B_A. Así, un método del servicio `S` en el *brick* B_A se invocará como:

```
proxyS.metodo (parámetros)
```

Obsérvese que en `addExternalService` no hay ningún parámetro que obligue al *brick* cliente a indicar cuál es el *brick* servidor: el *brick* cliente sólo conoce la interfaz del servicio. Sin embargo, dentro de la arquitectura software se tendrá que saber quién es el *brick* servidor para hacerle llegar las peticiones. En el estilo C2, este es un problema de encaminamiento que deben resolver los conectores. A continuación se verán varias soluciones a este problema a través del estudio de las políticas de filtrado del estilo C2 ofrecidas por *ArchStudio 3.0*.

4.5 Conectores C2 en *ArchStudio 3.0*

En *ArchStudio 3.0*, un conector C2 es una especialización de la clase `c2.legacy.AbstractC2DelegateBrick`, al que se le dota de un tratamiento especializado de mensajes de acuerdo a la política de filtrado que implemente. *ArchStudio 3.0* define dos tipos de conectores: el conector `BusConnector`, que implementa la política *no filtering* y el conector `FilteringBusConnector`, que implementa una política de filtrado punto a punto parecida a la política *message filtering*.

El conector `c2.legacy.conn.BusConnector`. Se encarga de hacer *broadcast* de cada mensaje que recibe, enviándolo a todos los componentes conectados en la interfaz opuesta por la que se recibió el mensaje. Aunque pueda ser ineficiente, el problema planteado en el apartado anterior de la localización del *brick* servidor está solucionado ya que seguro que llegará al componente servidor.

El conector `c2.legacy.conn.FilteringBusConnector`. Se usa únicamente cuando hay componentes que ofrecen servicios al resto de los componentes mediante el mecanismo de llamadas a procedimiento. El funcionamiento de este conector se ilustrará mediante un ejemplo: supóngase que dos componentes C_A y C_B exportan la misma interfaz de servicio `interfaz`, y el componente C_D desea usar los métodos de dicha interfaz (fig. 4). Cuando los componentes C_A y C_B se inicializan, envían un mensaje al conector para indicarle la interfaz que exportan, tal como se ve en el paso 1. El conector `Filtering Connector`$_1$ anota, en el paso 2, los pares (C_A, `interfaz`) y (C_B, `interfaz`). En el paso 3, el componente C_D envía un mensaje a `Filtering Connector`$_1$ solicitando un servicio de la interfaz. Cuando llega ese mensaje al conector, éste busca en su tabla el primer componente que soporte dicha interfaz. El primero que encuentra es C_A, por lo que le reenvía el mensaje de petición de C_D en el paso 4. Finalmente, en el paso 5, C_A envía la respuesta directamente a C_D sin pasar por el conector. Se observa que este conector no implementa la política *message filtering* por dos razones. La primera es relativa al mensaje que envían los componentes C_A y C_B para indicar que exportan su interfaz. Este mensaje, después de que el conector hubiese anotado las interfaces exportadas, debería propagarlo hacia los conectores conectados en la parte inferior (`Filtering Connector`$_2$), para hacerlo público al resto de la arquitectura. La segunda razón hace referencia a que si dos componentes ofrecen la misma interfaz, como es el caso de C_A y C_B, el conector debería enviar a todos la petición. Sin embargo, como se puede ver en el paso 4, el conector sólo la envía a C_A.

De lo anteriormente expuesto se deduce que este conector no soporta la política *message filtering* tal y como se define en C2. Pero el aspecto más importante es que este conector permite la comunicación directa entre dos componentes C_A y C_D como se observa en el paso 5. *ArchStudio 3.0* está construido con el estilo arquitectónico C2 que prohíbe este tipo de comunicación. Se preguntó a los autores de *ArchStudio 3.0* por el comportamiento de este filtro y la justificación que aportaron era que mejoraba

el rendimiento del sistema. Nosotros pensamos que el rendimiento se puede mejorar reduciendo el número de mensajes mediante el uso de conectores *message filtering* que envían mensajes únicamente a los componentes que pueden entenderlo. En el apartado siguiente se describirá el algoritmo que proponemos para la política *message filtering*.

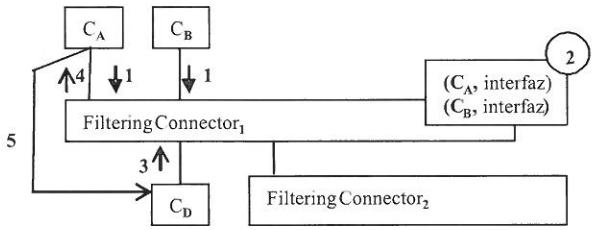

Fig. 4. Parte de una topología C2 con un conector de tipo *FilteringBusConnector*

5. Mejoras a *ArchStudio 3.0*: política de filtrado *message filtering*

Sobre *ArchStudio 3.0* se han realizado varias mejoras. La primera de ellas hace referencia a correcciones sobre el código fuente. En este sentido se ha comunicado a los autores de *ArchStudio 3.0* diferentes errores que hemos detectado, como código redundante o ligeros errores de funcionamiento. Estas modificaciones se incorporarán en la siguiente versión de la herramienta. La segunda mejora hace referencia a la propia documentación de la herramienta de la que este trabajo es un breve resumen. Finalmente, la mejora más importante es la incorporación, por nuestra parte, de la política de filtrado *message filtering* tal y como se define en C2. En la siguiente sección se comenta el algoritmo que hemos diseñado para soportar dicha política.

5.1 Algoritmo propuesto para soportar la política de filtrado *message filtering*

El algoritmo que se propone tiene dos objetivos. El objetivo principal es implementar fielmente (según está descrito por C2) la política de filtrado *message filtering*. El objetivo secundario es incluir este algoritmo en *ArchStudio 3.0* realizando en éste las modificaciones mínimas. La clave del algoritmo es cómo hacer llegar un *request* de una interfaz a todos los componentes servidores que entienden dicha interfaz (y sólo a ellos) y cómo hacer que las *notifications* que generan estos componentes servidores, como respuesta al *request* previo, sólo lleguen al componente cliente que envió dicho *request*. Es decir, la clave está en cómo obtener y mantener la información de encaminamiento necesaria para el envío de *requests* y la recepción de *notifications*. El algoritmo se va a describir en función del tratamiento que un conector dará a cada uno de los tres tipos de mensajes que debe procesar, a saber:

- un mensaje $notification_{ExpSer}$, que indica que se exporta un servicio,

- un mensaje *request* para solicitar un servicio,
- un mensaje *notification* que es contestación de un *request* previo,

A continuación se describe el tratamiento que el conector da a cada uno de ellos.

Tratamiento de un mensaje de tipo *Notification* que exporta un servicio. Una *notification* de exportación de servicio, $notification_{ExpSer}$, es un mensaje que un componente envía al conector, al que está conectado en su interfaz *bottom,* para indicarle que exporta un servicio. Esta *notification* se produce siempre que un componente se da de alta en tiempo de configuración de la arquitectura, o bien en tiempo de reconfiguración dinámica de ésta. Esta $notification_{ExpSer}$ contendrá tres campos principales: *source*, que indica el componente que exporta el servicio; *destination*, que indica el conector al que está conectado el componente en su interfaz *bottom*; e *interface*, que es el nombre de la interfaz que exporta el componente. Si el conector, que recibe dicha $notification_{ExpSer}$, es *no filtering,* hará *broadcast* de ella a todos los conectores y componentes conectados a su interfaz *bottom*, haciendo que en dichas *notifications* figure como *source* el propio conector *no filtering*. Si el mensaje $Notification_{ExpSer}$ llega a un conector de tipo *message filtering*, éste extraerá de la *notification* la información de encaminamiento relativa a la interfaz que se está exportando, para que, posteriormente, sepa a dónde enviar las *requests* que le lleguen sobre esta interfaz. Esta información la guardará el conector en su tabla local *Interfaces* en forma de par ($Notification_{ExpSer}.interface$, $Notification_{ExpSer}.source$). Es decir, guardará el nombre de la interfaz exportada y el emisor de dicha notificación. Este último será el componente que exporta la interfaz o un conector de cualquier tipo, que está propagando dicha exportación hacia los conectores situados en niveles inferiores de la arquitectura. Cuando termine el proceso de inicialización de los componentes, la tabla *Interfaces* del conector *message filtering* guardará el conjunto de todas las interfaces exportadas por los componentes conectados directa o indirectamente (a través de otros conectores) a la interfaz *top* de dicho conector y el conjunto de los elementos, componentes o conectores, conectados directamente al conector *message filtering*, a los que dicho conector tendrá que redirigir las *requests* relativas a estas interfaces. Esta información de encaminamiento es necesaria ya que, por una parte, en un *request* se indica la interfaz pero no el componente o componentes que soportan la interfaz y, por otra, el conector sólo conoce a los conectores o componentes que están conectados directamente a él, pero no conoce nada más de la topología. Finalmente, después de que el conector *message filtering* ha apuntado la información (interfaz, elemento de encaminamiento) en la tabla *Interfaces*, hace *broadcast* de la *notification* de exportación de servicio $Notification_{ExpSer}$, a todos los conectores conectados con él por su interfaz *bottom*, poniéndose como emisor de dicha notificación para que el resto de conectores inferiores de tipo *message filtering* tomen nota en sus tablas de las interfaces que pueden resolver.

En la fig.5 se puede ver un ejemplo de exportación de servicio. El conector $conn_2$ recibe una *notification* de exportación de servicio $interface_i$, procedente de un componente C_A (mensaje 1), y de otro conector $conn_1$, (mensaje 2) que está propagando el servicio $interface_j$ exportado por otro componente C_B. Cuando el

mensaje 1, llega a $conn_2$, éste apunta en su tabla *Interfaces* el par *(interface$_i$, C_A)*. En el caso del mensaje 2, el conector $conn_1$, apunta en su tabla *Interfaces (interface$_j$, C_B)* y después hace *broadcast* a los conectores, conectados directamente a él, por la interfaz *bottom*, lo que origina que $conn_2$ apunte en su tabla *Interfaces (interface$_j$, conn1)*. Obsérvese que $conn_2$ apunta en su tabla, como emisor a $conn_1$ en lugar de C_B puesto que sólo tiene comunicación directa con $conn_1$ y no con C_B.

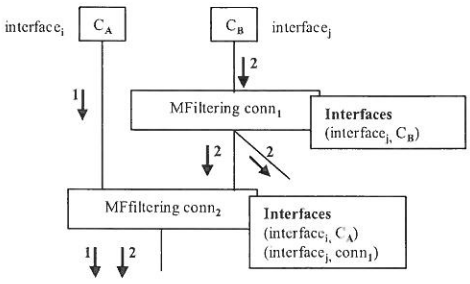

Fig. 5. Notificaciones para exportar interfaces

Tratamiento de un mensaje de tipo *Request*. Si el objetivo de un conector *message filtering* es hacer llegar el *request* sólo a los componentes que lo entiendan, el otro objetivo es hacer que las *notifications,* que se generen como respuesta a dicha petición, lleguen sólo al componente cliente que emitió el *request*. El primer objetivo se ha conseguido mediante el uso de tablas locales de interfaces dentro de cada conector *message filtering*, que registran los interfaces que cada conector es capaz de resolver parcialmente. Así, cuando le llegue un *request* lo reenviará a los elementos (componentes o conectores) que entiendan la interfaz solicitada en el *request*. El segundo objetivo se va a conseguir incluyendo en el *request* el identificador de cada conector que atraviesa el *request*, en un nuevo campo éste que llamaremos *path*.

A medida que el *request* atraviesa un conector *message filtering,* éste irá apilando un nuevo campo *path$_i$* con su identificador. Además, para facilitar la gestión de los campos de tipo *path* se incluirá un campo *numPath* que irá indicando cuántos campos de este tipo existen en el *request*. Cuando el componente servidor genere la *notification* de respuesta, estos campos *path* deberán figurar en dicha *notification* de tal forma que, cada vez que se atraviese un conector *message filtering$_i$,* se irá desapilando el campo *path$_i$* para saber por dónde enviar la *notification*. La gestión de los campos *path* sólo debe hacerse por parte de los conectores *message filtering* que son los encargados del encaminamiento; es decir, está información debe ser transparente al componente que emite el *request* y a los conectores *no filtering* que hacen *broadcast* del *request*. Esto obliga a tratar este caso particular: cuando a un conector *message filtering$_A$* le llegue un *request* directamente del componente cliente o de un conector *no filtering* deberá incluir dos campos *path*: el primero correspondiente al componente o conector *no filtering* emisor del *request* y el segundo correspondiente al propio conector *message filtering$_A$*. En cualquier otro caso, el mensaje provendrá de un conector *message filtering$_B$*, por lo que sólo tendrá

que apilar el identificador *message filtering*$_A$. Para ilustrar el mecanismo descrito anteriormente, se mostrarán los mensajes generados como consecuencia de la petición que el componente C_C emite sobre el *método*$_k$ de la interfaz *interface*$_j$ del componente C_E (fig. 6).

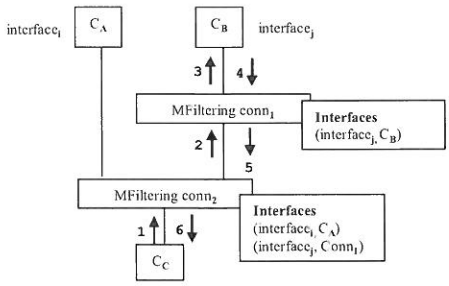

Fig. 6. *Request* de C_c sobre el interface$_j$

Supóngase, además, que los conectores usados, conn$_1$ y conn$_2$, son de tipo *message filtering*. El *request* inicial que el componente C_c envía por su interfaz *top* al conector conn$_2$, (mensaje 1) contendrá la información relativa a la interfaz, método, tipos de parámetros y valor de cada uno de ellos, como se muestra en la fig. 7:

Source	Dest.	Parameter	Parameter	Parameter*	Parameter*
C_C Top	conn$_2$ Bottom	("targetInterface", interface$_j$)	("MethodName", method$_k$)	("ParamType", parameterType$_i$)	("ParamValue", parameterValue$_i$)

Fig. 7. *Request* que C_c envía al conector conn$_2$

En la descripción del formato de dicho *request* se han sombreado los campos con la información de la interfaz, método y parámetros de la petición. Éstos estarán siempre en todos los *requests* posteriores que se generen, pero por simplificar, en adelante sólo se mostrará el parámetro que nombra la interfaz.

El *request* generado por el conector conn$_2$ (mensaje 2), se enviará al conector conn$_1$, ya que en la tabla *Interfaces* de conn$_2$, conn$_1$ es el único que entiende la *interface*$_j$. En el nuevo *request* (fig. 8), se añadirán dos nuevos parámetros *path*: uno con el componente C_C, que generó el *request* previo y otro con el identificador del conector actual conn$_2$ que genera este *request*.

El *request* generado por el conector conn$_1$ (mensaje 3), se enviará al componente C_B, ya que en la tabla *Interfaces* de conn$_1$ es el único que entiende la interfaz *interface*$_j$. Además, añadirá un nuevo parámetro *path* con el identificador del emisor del *request* 2 (fig. 9). Finalmente este *request* llegará al componente C_B, que servirá la petición, y construirá una *notification* de respuesta con el resultado del servicio solicitado junto con la información de encaminamiento contenida en el *request* previo.

Source	Dest.	Parameter	...	Parameter	Parameter	Parameter
$conn_2$ Top	$conn_1$ Bottom	("targetInterface", $interface_i$)		("NumPath", 2)	("Path1", C_CTop)	("Path2", $conn_2$Top)

Fig. 8. *Request* que el conector $conn_2$ envía al conector $conn_1$

Source	Dest.	Parameter	...	Parameter	Parameter	Parameter	Parameter
$conn_1$ Top	C_B Bottom	("targetInterface", $interface_i$)		("NumPath", 3)	("Path1", C_CTop)	("Path2", $conn_2$Top)	("Path3", $conn_1$Top)

Fig. 9. *Request* que el conector $conn_1$ envía a C_B

Tratamiento de un mensaje de tipo *Notification*. Se supone que un mensaje de tipo *notification*, generado como respuesta a un *request* previo, contendrá los parámetros de tipo *path* que se incluyeron en el *request*. Esta *notification* llegará a un conector *message filtering* por su interfaz *top*. El conector desapilará el último parámetro *path* de la *notification,* lo que supone decrementar también en 1 el valor del campo *numPath*. Si este parámetro no coincide con el identificador del conector actual, descarta el mensaje. Si coincide con su identificador entonces es que el *request* previo pasó por este conector *message filtering*. Ahora, en la cima de la pila de los parámetros *path* de la *notification* queda el identificador del elemento que le envió a este conector el *request* previo. Este elemento podrá ser: (1) un conector *message filtering, (2)* un conector *no filtering* o (3) el componente cliente que realizó la petición. En el caso (1) le envía la *notification* con la cima actualizada, poniendo en el campo *source* de la nueva *notification* el identificador del conector actual. En los casos (2) y (3) el conector desapila de nuevo la cima y decrementa otra vez *numPath*, ya que, como ocurría al generar el *request,* estos elementos no tratan información de encaminamiento. Tras actualizar la cima de *paths* envía la *notification* al elemento recién desapilado. Obsérvese que éste es un caso particular en donde se han desapilado dos elementos *path*. Este proceso de encaminamiento se irá repitiendo a medida que se atraviesen conectores *message filtering* hasta que la *notification* llegue al componente que generó la *request* previa. Volviendo al ejemplo de la fig. 6, una vez que el componente servidor (C_B) ha ejecutado el servicio solicitado, genera una *notification* (fig. 10) con el resultado (mensaje 4), con destino al conector $conn_1$ y cuyo contenido será el siguiente:

Source	Dest.	Parameter	...	Parameter	Parameter	Parameter	Parameter
C_B Bottom	$conn_1$ Top	("resulType", resulValue)		("NumPath", 3)	("Path1", C_CTop)	("Path2", $conn_2$Top)	("Path3", $conn_1$Top)

Fig. 10. *Notification* que envía C_B al conector $conn_1$

Los campos sombreados engloban toda la información relativa al resultado de la ejecución de método que, por simplificar, no se muestran. Cuando el conector $conn_1$

desapila de la *notification* (fig. 10) el último elemento *path3* con valor $conn_1$, descubre que efectivamente el *request* previo pasó por él. Por tanto, decrementa *NumPath* en 1 y comprueba que el elemento *path2* es un conector *message filtering*. Puesto que sí es un conector de tipo *message filtering*, reenvía la notificación que se indica en la figura 11.

Source	Dest.	Parameter	...	Parameter	Parameter	Parameter
$conn_1$ Bottom	$Conn_2$ Top	("resulType", resulValue)		("NumPath", 2)	("Path1", C_cTop)	("Path2", $conn_2$Top)

Fig. 11. *Notification* que el conector $conn_1$ manda al conector $conn_2$

Cuando el conector conn2 recibe la notificación (mensaje 5), éste desapila el elemento que ocupa la cima de la pila, *path2*, cuyo valor es $conn_2$, lo que indica que el *request* que ha originado esta *notification* ha pasado por el conector $conn_2$. Después, $conn_2$ comprueba si el elemento que ocupa ahora la cima, path1, es un conector *no filtering* o un componente. Como es un componente, $conn_2$ desapila también $path_1$, y elimina el campo *NumPath* puesto que la pila está vacía. Después reenvía la *notification* al componente C_c. Finalmente, el componente C_c obtiene la *notification* (mensaje 6) que es la respuesta al *request* que envió (mensaje 1), como se muestra en la fig. 12.

Source	Dest.	Parameter	...
$conn_2$ Bottom	C_c Top	("resulType", resulValue)	

Fig. 12. *Notification* que el conector $conn_1$ manda al componente C_c

6. Conclusiones y trabajos futuros

Del estudio de *ArchStudio 3.0* se han obtenido varios resultados en relación con la propia arquitectura de la herramienta y en relación al soporte que esta herramienta proporciona al estilo C2. Quizás, su mayor virtud es la flexibilidad que ofrece para configurar tanto la implementación de su máquina virtual como los elementos o *bricks* de la arquitectura software que se ejecutará sobre dicha máquina. Con respecto a los gestores que conforman su máquina virtual, el motor de la arquitectura permite diferentes políticas de *threading* para soportar los *bricks*, el gestor de comunicaciones puede apoyarse en uno o varios buzones para soportar el mecanismo de paso de mensajes y por último, el gestor de la arquitectura software permite definir arquitecturas siguiendo diferentes estilos arquitectónicos. En cuanto a los *bricks* de una arquitectura software, se pueden establecer fácilmente diferentes tipos de procesamiento de mensajes y diferentes políticas de *threading* para el tratamiento de mensajes. Además, la herramienta permite la personalización de prácticamente todas las funciones básicas, ya que éstas se definen mediante interfaces. No obstante,

ArchStudio 3.0 presenta algunas deficiencias. En primer lugar, con respecto al diseño de su máquina virtual, y que según los autores sigue el estilo C2, es discutible la utilización de un mecanismo basado en eventos para implementar el paso de mensajes ya que supone la existencia de memoria compartida, cuando el estilo C2 explícitamente indica lo contrario. En segundo lugar, y en cuanto al soporte del estilo C2, la deficiencia más importante encontrada es que las políticas de filtrado ofrecidas para los conectores no siguen la semántica del estilo. Su justificación, según los autores, está en la mejora del rendimiento al reducir el número de elementos que intervienen en la comunicación. Nosotros pensamos que el rendimiento se mejora si se reduce el número de mensajes, usando la política *message filtering*, en lugar de reducir el número de elementos usando política *no filtering*. Además, con esta estrategia no se violentan las reglas del estilo arquitectónico. En tercer lugar, la semántica de C2 sobre los conectores está incompleta, ya que no están implementadas las políticas de *message filtering*, *notification filtering*, *prioritized* y *sink*. En este sentido, en este trabajo hemos propuesto un algoritmo para implementar la política *message filtering* y la hemos incorporado a *ArchStudio 3.0*. Por último, indicar la carencia de documentación sobre la estructura interna de *ArchStudio 3.0* y que este trabajo viene a suplir.

En cuanto a los trabajos futuros, a corto plazo se pretende ampliar la documentación de la versión actual de *ArchStudio 3.0* para, a continuación, proceder al rediseño arquitectónico del mismo. A más largo plazo, nos planteamos dotar a la herramienta de las políticas de filtrado para los conectores C2 así como proveerla de una interfaz gráfica para realizar la descripción arquitectónica que, actualmente, se realiza de forma textual en *xADL 2.0*. Todo ello facilitará el diseño y ejecución de una arquitectura de componentes distribuidos para tolerancia a fallos.

Referencias

1. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley, Reading, MA, USA (1998).
2. Birrell, A. and Nelson, B. (1984): "Implementing Remote Procedure Calls." ACM Transactions on Computer Systems, vol. 2, no. 1, pp. 39-59.
3. ISR University of California, Irvine. http://www.isr.uci.edu/projects/ArchStudio.
4. Medvidovic, N.: Architecture-based Specification-time Software Evolution. Doctoral Dissertation, University of California, Irvine (1999).
5. OMG.: Model Driven Architecture (MDA). Document number ormsc/2001-07-01, Architectural Board ORMSC. (2001).
6. Shaw, M., Garlan, D.: Software Architecture. Perspectives on an Emerging Discipline. Prentice-Hall, NJ, USA (1996).
7. Soriano Salvador, E., Muñoz Fernández, I., Pérez Martínez, Jorge E.: Infraestructura para la Comunicación entre Componentes Java en el Estilo Arquitectónico C2. XII Jornadas de Concurrencia y Distribuidos. Avila, Spain (2004).